ELSEVIER

Science of Computer Programming 38 (2000) 125–142

Science of Computer Programming

www.elsevier.nl/locate/scico

machines

Stephan Diehl *

FB-14 Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, Germany

Received 6 September 1999

Abstract

In this paper we demonstrate how to use a semantics-directed generator to systematically design abstract machines. The main novelty of the generator is that it generates compilers *and* abstract machines. The generator is fully automated and its core transformations are proved correct. In this paper we propose a design methodology based on our generator and as an example we design a functional abstract machine which turns out to be very similar to the categorial abstract machine. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Abstract machines; Natural semantics; Compiler generation

1. Introduction

Abstract machines provide intermediate target languages for compilation. First, the compiler generates code for the abstract machine. Then, this code can be interpreted or further compiled into real-machine code. Abstract machines as an intermediate stage increase portability and maintainability of compilers. The instructions of an abstract machine are tailored to specific operations required to implement operations of a source language. For almost all kinds of languages, there exist abstract machines, e.g. for imperative [32], functional [4,5,17,21], logic [2,35], functional/logic [30], constraint [16], concurrent constraint [26] or object oriented [3,34] languages.

Abstract machines are usually designed in an ad hoc manner often based on experience with other abstract machines. But also some systematic approaches have been investigated. One of those is based on partial evaluation of example programs [9,20,31]. Another approach is to use pass separation transformations [18]. Hannan [15] introduced a pass separation transformation, which splits a set of term rewriting rules representing an abstract interpreter into two sets of term rewriting rules: the first set represents a

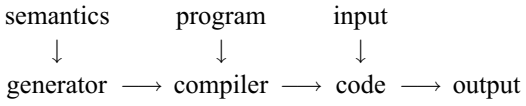
* Tel.: +49-681-302-3915.
E-mail address: diehl@cs.uni-sb.de (S. Diehl).

PII: S0167-6423(99)00047-7

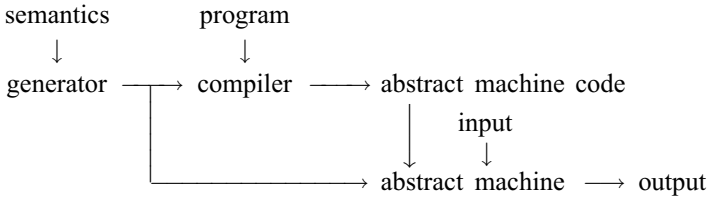
compiler into an abstract machine language, while the second set represents an abstract machine. We combined this transformation with other transformations, including extensions of those suggested by McKeever [25] and daSilva [13] and implemented all transformations in Prolog. As a result we get a system which given a natural semantics specification of a source language generates specifications of a compiler and an abstract machine for that source language. The generated specifications are term rewriting systems and our system is able to interprete these rules or translate them into SML or C.

For the formal definition of our meta-language and the transformations see [12], for the correctness proofs see [10].

Given a semantics specification of a source language, current semantics-directed compiler generators produce compilers from the source language into a fixed target language.



Rather than just generating compilers which translate source programs into a fixed target language, our system generates both a compiler and an abstract machine. The generated compiler translates source programs into code for the abstract machine.



In the next section we will present a methodology which uses the above generator in an iterative design process for abstract machines. Then we will demonstrate the methodology by means of an example: We will design an abstract machine for a simple functional language.

2. A generative methodology

Several authors [9,20,31] suggested a methodology based on partial evaluation, which consists of the following steps:

(1) Define an interpreter.
(2) Change the interpreter by making the heap representation and the unifications explicit.
(3) Partially evaluate the interpreter with respect to some example inputs.

(4) Look for patterns in the intermediate code and define corresponding machine instructions.
(5) Fold the result of the partial evaluation using the machine instructions.
(6) If the resulting code contains predicates or Prolog constructs, which are not machine instructions, or the machine instructions are not deterministic or not suitable for conventional computer architectures, then repeat the design loop (steps 2–6).

The problem with this approach is, that it does not guarantee that the designed compiler and abstract machine are complete. By complete, we mean that the generated compiler is able to translate every correct program into abstract machine code. The incompleteness is due to the fact that the design is based only on example programs.

In comparison our generator guarantees completeness, as it transforms semantics specifications and not example programs. In practice, the design of an abstract machine is a typical design process as we know it from software engineering: First, a prototype is constructed. Then, it is modified based on experience gained with the prototype. The modifications are done in the implementation of the abstract machine and the compiler has to be adapted. With the help of our generator, this process can be facilitated by lifting the modifications into the semantics specifications. This is advantageous, as correctness proofs of the modifications can be done on the semantics level. Thus our new methodology consists of the following steps:

(1) Define the semantics of the source language.
(2) Generate a compiler and abstract machine.
(3) Look for inefficiencies in the intermediate code and the definitions of the abstract machine instructions.
(4) Modify the semantics, then repeat the design loop (steps 2–4).

Modifications include that implementation details of data types are made explicit, e.g. the heap representation, or that data with different binding times are separated, this is also known as binding-time improvement.

3. Semantics notation: 2BIG

Natural semantics has been used by programming language researchers to specify many aspects of programming languages [7,8,16,19,27,28,33]. In this article we use natural semantics to specify the dynamic semantics of programming languages. But first we introduce our language to write natural semantics specifications: 2BIG

The language combines the structural approach of natural semantics [19] and the separation of general and implementation details by the use of a separately given interpretation for function symbols.

We refer to the specification of the interpretation for function symbols as the second level and the inference rules as the first level. All transformations performed by the generator are syntactical transformations on the inference rules, the interpretations of

functions remain unchanged. All compile-time computations have to be in the first level, run-time aspects can be hidden in the second level. Only the names and signatures of the function symbols are made available to the first level.

In 2BIG the dynamic semantics of a programming language is defined by a set of inference rules, e.g., the semantics of an assignment instruction which binds a name X to a value V is specified below:

$$r = \frac{member((X \mapsto Y), S) \quad V \rhd S \to N}{assign(X, V) \rhd S \to replace(X, S, N)}.$$

We will use the notational convention that meta-variables $c, c', c_i, e, e', e_i, \ldots$ denote terms.[1] Judgements are transitions of the form $c \rhd e_1 \to e_2$, e.g. $assign(X, V) \rhd S \to replace(X, S, N)$, or side conditions of the form $p(t_1, \ldots, t_n)$ or *not* $p(t_1, \ldots, t_n)$, e.g., $member((X \mapsto Y), S)$. In a rule the judgements above the line are called *preconditions* and the judgement below the line is called the *conclusion*. Furthermore, we adopt the notation for list constructors from Prolog, e.g., $[1, 2, 3] = [1|[2|[3|[]]]]$.

Functions (e.g., *replace* and the characteristic function of *member*) are defined separately, e.g., as Prolog predicates. We refer to their definitions as the second level of the specification.

In a transition $c \rhd e \to e'$ the meta-variables c, e and e' denote terms, thus they are not different entities. But to emphasize, that c, e and e' play different roles in a transition, we call the term on the left of $\rhd$ the *instruction*, the terms on the right of $\rhd$ and $\to$ are called *states* and we will refer to the outermost constructor of an instruction as an *instruction symbol*. This convention is motivated by the way transitions are used in semantics specifications. Usually a transition of the form $c \rhd e_1 \to e_2$ is interpreted as "the execution of the instructions c in state e_1 yields state e_2". By instructions we mean the constructs of the language being defined and by state we mean run-time information like bindings, environments or stores. Some authors refer to instruction-state pairs as configurations.

As a deviation from most works in natural semantics we use $c \rhd e$ instead of $e \vdash c$. As the rules usually define different cases for c, not for e, we feel that our notation is more readable.

The semantics of 2BIG is based on inductive systems [1] and relational inductive systems [13,14]. The relevant definitions are given in [12]. After transformation of side conditions into transitions[2] the 2BIG rules are simply relational inductive rules with ordered premises where we regard $\to$ as a ternary constructor symbol.

2BIG rules can also be given a procedural reading as in logic programming [23]. Informally, to prove that a transition $c \rhd e_1 \to e_2$ (goal) follows from the inference

[1] We use uppercase letters for variables of the specification language and lowercase letters for meta-variables. The term *variable* is also often used for the concept of binding values to names in programming languages, e.g. in the above 2BIG rule the specification language variable X gets bound to names of variables of the source language.

[2] We regard side conditions as syntactic sugar, which can be transformed into transitions containing the characteristic functions of the predicates.

rules, we unify it with the conclusion of a rule. If it unifies then the preconditions of that rule become our new goals. If the rule has no preconditions, then the goal trivially follows from that rule. This procedural reading underlies the Prolog implementation of 2BIG.

4. A generator for compilers and abstract machines

Our generator applies a sequence of established techniques to a natural semantics specification in order to split it into a compiler and an abstract machine. We believe that our framework, by virtue of being compositional, can be extended over time to include even more powerful analysis and transformation methods. Actually, the transformations are mostly source to source and after every transformation we have an executable specification again. Of these transformations pass separation is the most important one. Let p be a program and x and y the static and dynamic input to this program, then partial evaluation of p with respect to x yields a residual program p_x, such that $p_x(y) = p(x, y)$. In contrast, pass separation transforms the program p into two programs p_1 and p_2 such that $p_2(p_1(x), y) = p(x, y)$. Note that here p_1 produces some intermediate data, which are input to p_2. When it comes to the generation of compiler/executor pairs, pass separation provides an immediate solution, we pass separate the interpreter *interp* into an executor *exec* and a compiler *comp*, such that: $interp(prog, data) = exec(comp(prog), data)$. Despite this potential for compiler generation there is only little work on pass separation [11,15,18].

Our generator first transforms the 2BIG rules into a term rewriting system:

For this, it first removes side conditions by converting them into transitions, thus there are now only transitions as preconditions. Then it factorizes rules which have a common initial sequence of preconditions. Factorization replaces these rules with a single rule which has the common initial sequence as its preconditions and for each original rule a rule is generated with its remaining preconditions. Next, the generator adds a stack to the state in the transitions and stores temporary variables, i.e. variables which are not used in an intermediate transition. Variables which do not occur in the conclusion of a rule are eliminated. The last step before the actual transformation into a term rewriting system is called sequentialization. It converts all preconditions of a rule such that the result state of one transition is the start state of the next. These rules can now be easily turned into rewrite rules. Rules of the form $(c_1 \rhd e_1 \to e_1' \ \cdots \ c_n \rhd e_n \to e_n')/c \rhd e \to e'$ are converted into $\langle (c; p), e\rangle \to \langle (c_1; \ldots; c_n; p), e_1\rangle$ where p is a new variable name.

Now the resulting term rewriting system is in a form, such that pass separation can be applied which yields two-term rewriting systems: one representing a compiler and one representing the abstract machine. These term rewriting systems are then further optimized to reduce the number and complexity of the abstract machine instructions, e.g. the number of arguments.

Informally, the relation of the generated compiler and abstract machine and the original 2BIG specification is as follows: If the transition $p \rhd e \to e'$ for a program p with start state e can be proven using the original 2BIG rules, then $\langle \tilde{p}, [[], \tilde{e}\rangle$ can be rewritten using the abstract machine rules into $\langle no\, p, [[], \tilde{e'}]\rangle$ where $\tilde{p}$, $\tilde{e}$ and $\tilde{e'}$ are the compiled versions of the terms p, e and e'.[3] Note, that also the states are compiled as we allow that source language constructs can occur in a state, e.g. the state might contain an environment mapping names to function abstraction as in the case of Mini-ML.

In the rest of the paper we will use the generator as a black box. Similar to other generation tools in compiler design like `lex` or `yacc`, it is based on a formal theory [12]. When we use these tools, we do not have to know much about their inner workings, but we have to understand their specification languages.

5. Designing an abstract machine for Mini-ML

We apply our system to a specification of Mini-ML. The generated compiler and abstract machine are similar to those presented in [15], i.e., the categorial abstract machine (CAM). The CAM has been the basis of very efficient implementations of ML [22,24]. Based on the specification of Mini-ML in [7,19] we will present a 2BIG specification of Mini-ML and the compiler and abstract machine generated by our system. A closer look at the abstract machine instructions reveals that variable lookup is still inefficient. We introduce an abstract syntax and a conversion of Mini-ML programs into the abstract syntax. In the abstract syntax variable names are replaced by access paths which are just a different encoding for deBruijn numerals. For this abstract syntax we give a 2BIG specification. It turns out that the generated abstract machine is close to an existing abstract machine, the CAM.

5.1. Mini-ML

In [7,19] the authors present natural semantics specifications of Mini-ML, the CAM and the translation of Mini-ML programs to CAM code. Mini-ML consists of the purely applicative part of ML, more precisely a simple typed λ-calculus with constants, pairs, conditionals and recursive function definitions. The syntax of Mini-ML is given in Fig. 1, its semantics will be defined in the next section by 2BIG rules.

Example 1. The following example program counts from 10 down to 0:

letrec $y = \lambda x.$**if** **equal**(**var**(x), **num**(0)) **then** **var**(x)
else (**var**(y) (**var**(x) − **num**(1)))
in (**var**(y) **num**(10)) **end**

[3] A similar inverse implication states that for every abstract machine execution of a compiled program there is a transition provable with the 2BIG rules.

x		variable symbol	
n		number	
b		boolean value: $true, false$	
E	::=	$\mathbf{bool}(b) \mid \mathbf{num}(n)$	boolean value, number
	|	$\mathbf{equal}(E, E)$	equality test
	|	$E + E \mid E - E$	sum, difference
	|	$\mathbf{var}(x)$	variable use
	|	**if** E **then** E **else** E **end**	conditional
	|	(E, E)	pair
	|	$\mathbf{fst}(E)$	first value of pair
	|	$\mathbf{snd}(E)$	second value of pair
	|	$\lambda x.E$	abstraction
	|	$(E\ E)$	function application
	|	**let** $x = E$ **in** E **end**	function definition
	|	**letrec** $x = E$ **in** E **end**	recursive function definition

Fig. 1. Syntax of Mini-ML.

And this is a Mini-ML function computing Fibonacci numbers:

letrec $fib = \lambda x.$**if equal**(**var**(x), **num**(0)) **then num**(0)
else if equal(**var**(x), **num**(1)) **then num**(1)
else (**var**(fib) (**var**(x) − **num**(1)))
+(**var**(fib) (**var**(x) − **num**(2)))
end
end
in (**var**(fib) **num**(10)) **end**

5.2. *Transforming a* 2BIG *specification of* Mini-ML

5.2.1. *Cyclic bindings*

The static semantics of our meta-language requires that the preconditions of a rule are well-ordered as defined in [12]. Well-orderedness does not allow for cyclic dependencies of variables and thus allows us to use terms and not graphs or rational trees as the underlying model of 2BIG rules. From a practical point of view it facilitates to generate term rewriting systems as specifications of abstract machines. To implement cyclic dependencies as for E^* in $(P_2 \rhd [[X \mapsto E^*]|E] \to E^* \quad P_1 \rhd [[X \mapsto E^*]|E] \to E')/($ **letrec** $X = P_2$ **in** P_1 **end** $\rhd E \to E')$ we lift the handling of cyclic bindings of variables in the meta-language into the specification, i.e., we specify the

indirection and dereferencing of variables by 2BIG rules. As a result the state in the 2BIG rules contains a new component, namely a list of redirections. Redirections associate indices (addresses) with values. Everytime a value is an index, we have to look up its value in the redirections. This process is also called dereferencing. Instead of the above rule for **letrec** we thus write a 2BIG using redirections:

$$\frac{\begin{array}{l}\textbf{newind} \rhd R \rightarrow N \\ P_2 \rhd [[\textbf{red}(N, \textbf{ind}(N))|R], \mathit{replace}(X, \textbf{ind}(N), E)] \rightarrow [R', E^*] \\ P_1 \rhd [\mathit{replace\_red}(N, E^*, R'), \mathit{replace}(X, \textbf{val}(E^*), E)] \rightarrow [R'', E']\end{array}}{\textbf{letrec } X = P_2 \textbf{ in } P_1 \textbf{ end} \rhd [R, E] \rightarrow [R'', E']}$$

Note, that here in the transition for P_2 the environment E^* only occurs once on the right-hand side. Thus, P_2 is executed in an environment where the redirection N is bound to $\mathit{ind}(N)$, whereas P_1 is executed in an environment where N is bound to $\textbf{val}(E^*)$.

5.2.2. Basic operations

In the 2BIG rules the following functions are used:

- $\mathit{lookup}(E, X)$ yields the value associated with the identifier bound to X in the mapping E.
- $\mathit{replace}(X, V, E)$ yields a new mapping, which differs from E only in that the association of the identifier X is replaced by an association of the identifier X to the value V.
- $\mathit{minus\_op}(V_1, V_2)$ yields $V_1 - V_2$ if both values are numbers.
- $\mathit{plus\_op}(V_1, V_2)$ yields $V_1 + V_2$ if both values are numbers.
- $\mathit{equal\_op}(V_1, V_2)$ yields *true* if both values are equal, *false* otherwise.
- $\mathit{lookup\_red}(R, N)$ yields the value associated with index N in the redirections R.
- $\mathit{replace\_red}(N, V, R)$ replaces the value associated with index N in the redirections R by the value V.
- $\mathit{new\_index}(R)$ yields a new index, which is not yet contained in R.

5.3. First shot

The 2BIG specification in Table 1 is based on the natural semantics specification of Mini-ML in [7,19]. To convert their natural semantics rules into 2BIG rules we had to remove the cyclic dependencies and enforce the static semantics of 2BIG.

We use constructors like **xbool**, **xnum** and **clo** to build intermediate data structures and distinguish them from constructors of the source language like **bool** or **num**. The constructor **val** indicates that a value has not to be dereferenced, whereas **ind** is an index and has to be dereferenced to get a value.

Table 1
First shot: natural semantics of Mini-ML

Primitive types:

$$\frac{}{\mathbf{num}(N) \rhd [R,E] \to [R, \mathbf{xnum}(N)]}$$

$$\frac{V_1 \rhd [R,E] \to [R', \mathbf{xnum}(N)] \qquad V_2 \rhd [R',E] \to [R'', \mathbf{xnum}(M)]}{V_1 + V_2 \rhd [R,E] \to [R'', \mathbf{xnum}(\mathit{plus\_op}(N,M))]}$$

$$\frac{V_1 \rhd [R,E] \to [R', \mathbf{xnum}(N)] \qquad V_2 \rhd [R',E] \to [R'', \mathbf{xnum}(M)]}{V_1 - V_2 \rhd [R,E] \to [R'', \mathbf{xnum}(\mathit{minus\_op}(N,M))]}$$

$$\frac{}{\mathbf{bool}(B) \rhd [R,E] \to [R, \mathbf{xbool}(B)]}$$

$$\frac{N \rhd [R,E] \to [R', N'] \qquad M \rhd [R',E] \to [R'', M']}{\mathbf{equal}(N,M) \rhd [R,E] \to [R'', \mathbf{xbool}(\mathit{equal\_op}(N', M'))]}$$

Pairs:

$$\frac{V_1 \rhd [R,E] \to [R', V_1'] \qquad V_2 \rhd [R',E] \to [R'', V_2']}{(V_1, V_2) \rhd [R,E] \to [R'', \mathbf{xpair}(V_1', V_2')]}$$

$$\frac{V \rhd [R,E] \to [R', \mathbf{xpair}(A,B)]}{\mathbf{fst}(V) \rhd [R,E] \to [R', A]} \qquad \frac{V \rhd [R,E] \to [R', \mathbf{xpair}(A,B)]}{\mathbf{snd}(V) \rhd [R,E] \to [R', B]}$$

Variable lookup:

$$\frac{\mathbf{lkup} \rhd [X,E] \to \mathbf{ind}(N)}{\mathbf{var}(X) \rhd [R,E] \to [R, \mathit{lookup\_red}(R,N)]} \qquad \frac{\mathbf{lkup} \rhd [X,E] \to \mathbf{val}(V)}{\mathbf{var}(X) \rhd [R,E] \to [R,V]}$$

$$\frac{}{\mathbf{lkup} \rhd [X,E] \to \mathit{lookup}(E,X)}$$

Conditional:

$$\frac{B \rhd [R,E] \to [R', \mathbf{xbool}(\mathbf{true})] \qquad V_1 \rhd [R',E] \to [R'', V_1']}{\mathbf{if}\ B\ \mathbf{then}\ V_1\ \mathbf{else}\ V_2\ \mathbf{end} \rhd [R,E] \to [R'', V_1']}$$

$$\frac{B \rhd [R,E] \to [R', \mathbf{xbool}(\mathbf{false})] \qquad V_2 \rhd [R',E] \to [R'', V_2']}{\mathbf{if}\ B\ \mathbf{then}\ V_1\ \mathbf{else}\ V_2\ \mathbf{end} \rhd [R,E] \to [R'', V_2']}$$

Functions:

$$\frac{}{\lambda X.V \rhd [R,E] \to [R, \mathbf{clo}(E, \mathbf{xlambda}(X,V))]}$$

$$\frac{V_1 \rhd [R,E] \to [R', \mathbf{clo}(E', \mathbf{xlambda}(X,C))] \qquad V_2 \rhd [R',E] \to [R'', V_2'] \qquad \mathbf{run} \rhd [C, R'', \mathit{replace}(X, \mathbf{val}(V_2'), E')] \to [R^*, V]}{(V_1\ V_2) \rhd [R,E] \to [R^*, V]}$$

$$\frac{C \rhd [R,E] \to [R', V]}{\mathbf{run} \rhd [C,R,E] \to [R', V]}$$

$$\frac{V_1 \rhd [R,E] \to [R', V_1'] \qquad V_2 \rhd [R', \mathit{replace}(X, \mathbf{val}(V_1'), E)] \to [R'', V]}{\mathbf{let}\ X = V_1\ \mathbf{in}\ V_2\ \mathbf{end} \rhd [R,E] \to [R'', V]}$$

Recursive functions:

$$\frac{\begin{array}{l}\mathbf{newind} \rhd R \to N \\ V_1 \rhd [[\mathbf{red}(N, \mathbf{ind}(N)) | R], \mathit{replace}(X, \mathbf{ind}(N), E)] \to [R', V_1'] \\ V_2 \rhd [\mathit{replace\_red}(N, V_1', R'), \mathit{replace}(X, \mathbf{val}(V_1'), E)] \to [R'', V_2']\end{array}}{\mathbf{letrec}\ X = V_1\ \mathbf{in}\ V_2\ \mathbf{end} \rhd [R,E] \to [R'', V_2']}$$

$$\frac{}{\mathbf{newind} \rhd R \to \mathit{new\_index}(R)}$$

5.3.1. Generated compiler and abstract machine for Mini-ML

Next, we apply our generator to the 2BIG specification in Table 1. It transforms the rules as described in Section 4 and finally produces the term rewriting rules for a compiler as shown in Table 2 and for an abstract machine as shown in Tables 3 and 4.

Table 2
First shot: generated compiler

$\mathbf{bool}(B)$	$\Rightarrow \overline{\mathbf{bool}}(B)$
$\mathbf{equal}(A,B)$	$\Rightarrow \overline{\mathbf{equal}}; A; \overline{\mathbf{conv}}\_\mathbf{0}; B; \overline{\mathbf{conv}}\_\mathbf{1}$
$\mathbf{num}(N)$	$\Rightarrow \overline{\mathbf{num}}(N)$
$A + B$	$\Rightarrow \overline{\mathbf{equal}}; A; \overline{\mathbf{conv}}\_\mathbf{2}; B; \overline{\mathbf{conv}}\_\mathbf{3}$
$A - B$	$\Rightarrow \overline{\mathbf{equal}}; A; \overline{\mathbf{conv}}\_\mathbf{2}; B; \overline{\mathbf{conv}}\_\mathbf{5}$
(A,B)	$\Rightarrow \overline{\mathbf{equal}}; A; \overline{\mathbf{conv}}\_\mathbf{0}; B; \overline{\mathbf{conv}}\_\mathbf{9}$
$\mathbf{fst}(V)$	$\Rightarrow \overline{\mathbf{fst}}; V; \overline{\mathbf{conv}}\_\mathbf{10}$
$\mathbf{snd}(V)$	$\Rightarrow \overline{\mathbf{fst}}; V; \overline{\mathbf{conv}}\_\mathbf{11}$
$\mathbf{var}(X)$	$\Rightarrow \overline{\mathbf{var}}(X)$
if B **then** V_1 **else** V_2 **end**	$\Rightarrow \overline{\mathbf{equal}}; B; \overline{\mathbf{conv}}\_\mathbf{7}; \overline{\mathbf{factor}}\_\mathbf{1}(V_2, V_1)$
$\lambda X.V$	$\Rightarrow \overline{\mathbf{lambda}}(X, V)$
$(V_1\ V_2)$	$\Rightarrow \overline{\mathbf{equal}}; V_1; \overline{\mathbf{conv}}\_\mathbf{12}; V_2; \overline{\mathbf{conv}}\_\mathbf{13}; \overline{\mathbf{run}}$
let $X = V_1$ **in** V_2 **end**	$\Rightarrow \overline{\mathbf{equal}}; V_1; \overline{\mathbf{conv}}\_\mathbf{15}(X); V_2$
letrec $X = V_1$ **in** V_2 **end**	$\Rightarrow \overline{\mathbf{letrec}}; \mathbf{newind}; \overline{\mathbf{conv}}\_\mathbf{16}(X); V_1; \overline{\mathbf{conv}}\_\mathbf{17}; V_2$

Table 3
AM_1: variable lookup

$\langle \overline{\mathbf{var}}(X); C,\ [D,[R,E]]\rangle$	$\Rightarrow$	$\langle \overline{\mathbf{lkup}}; \overline{\mathbf{conv}}\_\mathbf{6}; \overline{\mathbf{factor}}\_\mathbf{0}; C,\ [[[R]\|D],[X,E]]\rangle$
$\langle \overline{\mathbf{lkup}}; C,\ [D,[X,E]]\rangle$	$\Rightarrow$	$\langle C,\ [D, lookup(X,E)]\rangle$
$\langle \overline{\mathbf{conv}}\_\mathbf{6}; C,\ [[[R]\|D],S]\rangle$	$\Rightarrow$	$\langle C,\ [D,[[R],S]]\rangle$
$\langle \overline{\mathbf{factor}}\_\mathbf{0}; C,\ [D,[[R],\mathbf{ind}(N)]]\rangle$	$\Rightarrow$	$\langle C,\ [D,[R, lookup\_red(R,N)]]\rangle$
$\langle \overline{\mathbf{factor}}\_\mathbf{0}; C,\ [D,[[R],\mathbf{val}(V)]]\rangle$	$\Rightarrow$	$\langle C,\ [D,[R,V]]\rangle$

Table 4
AM_1: recursive functions

$\langle \overline{\mathbf{letrec}}; C,\ [D,[R,E]]\rangle \Rightarrow \langle C,\ [[[R,E]|D],R]\rangle$
$\langle \overline{\mathbf{newind}}; C,\ [D,R]\rangle \Rightarrow \langle C,\ [D, new\_index(R)]\rangle$
$\langle \overline{\mathbf{conv}}\_\mathbf{16}(X); C,\ [[[R,E]|D],N]\rangle \Rightarrow$
$\quad \langle C,\ [[[N,X,E]|D],[[\mathbf{red}(N,\mathbf{ind}(N))|R], replace(X,\mathbf{ind}(N),E)]]\rangle$
$\langle \overline{\mathbf{conv}}\_\mathbf{17}; C,\ [[[N,X,E]|D],[R,V]]\rangle \Rightarrow$
$\quad \langle C,\ [D,[replace\_red(N,V,R), replace(X,\mathbf{val}(V),E)]]\rangle$

The names of the instructions introduced by our system are $\overline{\mathbf{test}}\_\ldots$, $\overline{\mathbf{conv}}\_\ldots$, $\overline{\mathbf{factor}}\_\ldots$, $\overline{\mathbf{fact}}\_\ldots$ and $\overline{\mathbf{comb}}\_\ldots$. These names describe the task of an instruction not as specific as those instruction names we know from existing abstract machines. After a closer inspection of what our generated instructions do, we could give them more suggestive names like *push*, *pop*, *branch*, etc. For example the instruction $\overline{\mathbf{equal}}$ occurs quite often in the compiler rules. The name of the instruction was chosen, when the rule for the source language construct **equal** was generated. The instruction actually is only a push operation and does not do any comparison, so this is a case where the generated name is actually misleading. All later other occurrences of $\overline{\mathbf{equal}}$ in other

compiler rules are due to the fact that similar instructions have been generated for these rules, but because they had the same effects, they have been identified with **equal**.

Actually, Tables 3 and 4 contain only some of the generated abstract machine rules, as we only need these to illustrate and discuss some inefficiencies in the machine.

In the CAM the variable access is done by statically compiled access paths. In the above abstract machine rules variable access is still a search process hidden in the basic operation *lookup*. The instruction **factor_0** tests the cases that the variable is bound to a redirection or directly to a value.

Again the search process is hidden in a basic operation, namely the operations $replace(X, \mathbf{ind}(N), E)$ and $replace(X, \mathbf{val}(V), E)$. The association found for the variable X is then replaced by an association of X to the new value. In the CAM the environment is implemented as a stack and the new value is just put on top of the stack.

5.4. Second shot

As pointed out the structure of environments and as a result the variable lookup in the above abstract machine is still inefficient. Our transformations do not automatically change this structure, thus the 2BIG specification has to be modified. We introduce an abstract syntax with deBruijn numerals and a conversion of Mini-ML programs into the abstract syntax. Then we give a new 2BIG specification for Mini-ML programs in abstract syntax. When we replace variable names by deBruijn numerals the environment can be replaced by a stack. The value associated with the variable represented by the numeral \$0 is the top most element of the stack, for the numeral \$$n$ the value is the $(n-1)$th element of the stack. Access paths are a unary representation of numerals, i.e., the numeral \$0 is represented by **car**, the numeral \$$n$ by $\underbrace{\mathbf{cdr}(\mathbf{cdr}(\ldots\mathbf{cdr}}_{n \text{ times}}(\mathbf{car})))$.

5.4.1. Abstract syntax

Since we use now deBruijn numerals in λ, **let** and **letrec** abstractions, there are no more variable names in Mini-ML programs and we have to change the syntax (see Fig. 2).

V	::=	$car \mid cdr(V)$	variable access path
E	::=	...	
	|	V	variable use
	|	$\lambda.E$	abstraction
	|	**let** E **in** E **end**	function definition
	|	**letrec** E **in** E **end**	recursive function definition

Fig. 2. Abstract syntax of Mini-ML.

Example 2. Using the new abstract syntax the example program which counts from 10 down to 0 becomes

$$\begin{array}{l}\textbf{letrec}\ \lambda.\textbf{if}\ \textbf{equal}(\textbf{car},\textbf{num}(0))\ \textbf{then}\ \textbf{car}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (\textbf{cdr}(\textbf{car})\ (\textbf{car}-\textbf{num}(1)))\\ \textbf{in}\ (\textbf{car}\ \textbf{num}(10))\ \textbf{end}\end{array}$$

And the Mini-ML *function for Fibonacci numbers is now written as*:

$$\begin{array}{l}\textbf{letrec}\ \lambda.\textbf{if}\ \textbf{equal}(\textbf{car},\textbf{num}(0))\ \textbf{then}\ \textbf{num}(0)\\ \qquad\textbf{else if}\ \textbf{equal}(\textbf{car},\textbf{num}(1))\ \textbf{then}\ \textbf{num}(1)\\ \qquad\qquad\textbf{else}\ (\textbf{cdr}(\textbf{car})\ (\textbf{car}-\textbf{num}(1)))\\ \qquad\qquad\qquad +(\textbf{cdr}(\textbf{car})\ (\textbf{car}-\textbf{num}(2)))\\ \qquad\qquad\textbf{end}\\ \qquad\textbf{end}\\ \textbf{in}\ (\textbf{car}\ \textbf{num}(10))\ \textbf{end}\end{array}$$

5.4.2. Conversion into abstract syntax

$$\frac{}{\textbf{var}(X)\rhd E\to access\_path(X,E)} \tag{1}$$

Here $access\_path(X,E)$ is a basic operation which yields the access path for the numeral $\$(n-1)$ if X is the nth variable in E. Or in other words $access\_path(X,E) = \underbrace{\textbf{cdr}(\textbf{cdr}(\ldots\textbf{cdr}}_{n-1\ \ times}(\textbf{car})))$.

$$\begin{array}{c}\dfrac{V\rhd[X|E]\to V'}{\lambda X.V\rhd E\to\lambda.V'}\\[2ex] \dfrac{V_1\rhd E\to V_1'\quad V_2\rhd[X|E]\to V_2'}{\textbf{let}\ X=V_1\ \textbf{in}\ V_2\ \textbf{end}\rhd E\to\textbf{let}\ V_1'\ \textbf{in}\ V_2'\ \textbf{end}}\\[2ex] \dfrac{V_1\rhd[X|E]\to V_1'\quad V_2\rhd[X|E]\to V_2'}{\textbf{letrec}\ X=V_1\ \textbf{in}\ V_2\ \textbf{end}\rhd E\to\textbf{letrec}\ V_1'\ \textbf{in}\ V_2'\ \textbf{end}}\end{array} \tag{2}$$

In all other cases the arguments are just translated in the current environment, e.g.

$$\frac{V_1\rhd E\to V_1'\quad V_2\rhd E\to V_2'}{(V_1,V_2)\rhd E\to(V_1',V_2')} \tag{3}$$

2BIG *Specification of* Mini-ML

Variable lookup: Instead of rules for $\textbf{var}(X)$, we have now rules for access paths:

$$\begin{array}{c}\dfrac{}{\textbf{car}\rhd[R,[\textbf{ind}(M)|E]]\to[R,lookup\_red(R,M)]}\\[2ex] \dfrac{}{\textbf{car}\rhd[R,[\textbf{val}(V)|E]]\to[R,V]}\\[2ex] \dfrac{A\rhd[R,E]\to[R,V]}{\textbf{cdr}(A)\rhd[R,[H|E]]\to[R,V]}\end{array} \tag{4}$$

Functions: The name of the variable bound by the λ abstraction is no longer stored in the closure,

$$\frac{}{\lambda.C \rhd [R,E] \to [R, \mathbf{clo}(E, \mathbf{xlambda}(C))]} \tag{5}$$

Instead of replacing the value bound to the variable bound by the λ abstraction of the closure, we now pass the new value on top of the environment, which is now a stack and no longer a mapping of variable names to values:

$$\frac{\begin{array}{l} V_1 \rhd [R,E] \to [R', \mathbf{clo}(E', \mathbf{xlambda}(C))] \qquad V_2 \rhd [R',E] \to [R'', V_2'] \\ \mathbf{run} \rhd [C, R'', [\mathbf{val}(V_2')|E']] \to [R^*, V] \end{array}}{(V_1 \ V_2) \rhd [R,E] \to [R^*, V]} \tag{6}$$

Again, instead of binding the value V_1' to a variable it is passed on top of the stack,

$$\frac{V_1 \rhd [R,E] \to [R', V_1'] \qquad V_2 \rhd [R', [\mathbf{val}(V_1')|E]] \to [R'', V_2']}{\mathbf{let}\ V_1\ \mathbf{in}\ V_2\ \mathbf{end} \rhd [R,E] \to [R'', V_2']} \tag{7}$$

Recursive functions: And again, the value of V_1' is now passed on top of the stack,

$$\frac{\begin{array}{l} \mathbf{newind} \rhd R \to M \qquad V_1 \rhd [[\mathbf{red}(M, \mathbf{ind}(M))|R], [\mathbf{ind}(M)|E]] \to [R', V_1'] \\ V_2 \rhd [replace\_red(M, V_1', R'), [\mathbf{val}(V_1')|E]] \to [R'', V_2'] \end{array}}{\mathbf{letrec}\ V_1\ \mathbf{in}\ V_2\ \mathbf{end} \rhd [R,E] \to [R'', V_2']} \tag{8}$$

5.4.3. *Generated compiler and abstract machine for* Mini-ML

After modifying the 2BIG specification of Mini-ML as described above we apply the generator again and get the compiler rules in Table 5 and the abstract machine rules in Tables 6–8.

Access paths are now translated into sequences of instructions, thus the path **cdr**(**cdr**(**car**)) becomes $\overline{\mathbf{cdr}};\overline{\mathbf{cdr}};\overline{\mathbf{car}}$. Of course, there are no more abstract machine instructions, which take variable names as their arguments, e.g., X in $\overline{\mathbf{lambda}}(X, C)$.

Table 5
Second shot: generated compiler

car	$\Rightarrow \overline{\mathbf{car}}$
cdr(A)	$\Rightarrow \overline{\mathbf{cdr}}; A$
$\lambda.C$	$\Rightarrow \overline{\mathbf{lambda}}(C)$
$(V_1\ V_2)$	$\Rightarrow \overline{\mathbf{equal}}; V_1; \overline{\mathbf{conv}\_\mathbf{11}}; V_2; \overline{\mathbf{conv}\_\mathbf{12}}; \overline{\mathbf{run}}$
let V_1 **in** V_2 **end**	$\Rightarrow \overline{\mathbf{equal}}; V_1; \overline{\mathbf{conv}\_\mathbf{14}}; V_2$
letrec V_1 **in** V_2 **end**	$\Rightarrow \overline{\mathbf{letrec}}; \overline{\mathbf{newind}}; \overline{\mathbf{conv}\_\mathbf{15}}; V_1; \overline{\mathbf{conv}\_\mathbf{16}}; V_2$

Table 6
AM_2: variable lookup

$\langle \overline{\mathbf{car}}; C, [D, [R, [\mathbf{ind}(M)\|E]]] \rangle$	$\Rightarrow \langle C, [D, [R, lookup\_red(R, M)]] \rangle$
$\langle \overline{\mathbf{car}}; C, [D, [R, [\mathbf{val}(V)\|E]]] \rangle$	$\Rightarrow \langle C, [D, [R, V]] \rangle$
$\langle \overline{\mathbf{cdr}}; C, [D, [R, [V\|E]]] \rangle$	$\Rightarrow \langle C, [D, [R, E]] \rangle$

Table 7
AM_2: functions

$\langle \overline{\textbf{lambda}}(V); C,\ [D,[R,E]]\rangle \Rightarrow \langle C,\ [D,[R,\textbf{clo}(E,\textbf{xlambda}(V))]]\rangle$
$\langle \overline{\textbf{equal}}; C,\ [D,[R,E]]\rangle \Rightarrow \langle C,\ [[[E]|D],[R,E]]\rangle$
$\langle \overline{\textbf{conv\_11}}; C,\ [[[E]|D],[R,\textbf{clo}(E',\textbf{xlambda}(T))]]\rangle \Rightarrow$
$\langle C,\ [[[E',T]|D],[R,E]]\rangle$
$\langle \overline{\textbf{conv\_12}}; C,\ [[[E,T]|D],[R,V]]\rangle \Rightarrow \langle C,\ [D,[T,R,[\textbf{val}(V)|E]]]\rangle$
$\langle \overline{\textbf{run}}; C,\ [D,[T,R,E]]\rangle \Rightarrow \langle T; \overline{\textbf{conv\_13}}; C,\ [[[T]|D],[R,E]]\rangle$
$\langle \overline{\textbf{conv\_13}}; C,\ [[[T]|D],[R,E]]\rangle \Rightarrow \langle C,\ [D,[R,E]]\rangle$
$\langle \overline{\textbf{conv\_14}}; C,\ [[[E]|D],[R,V]]\rangle \Rightarrow \langle C,\ [D,[R,[\textbf{val}(V)|E]]]\rangle$

Table 8
AM_2: recursive functions

$\langle \overline{\textbf{letrec}}; C,\ [D,[R,E]]\rangle \Rightarrow \langle C,\ [[[R,E]|D],R]\rangle$
$\langle \overline{\textbf{conv\_15}}; C,\ [[[R,E]|D],M]\rangle \Rightarrow$
$\langle C,\ [[[M,E]|D],[[\textbf{red}(M,\textbf{ind}(M))|R],[\textbf{ind}(M)|E]]]\rangle$
$\langle \overline{\textbf{conv\_16}}; C,\ [[[M,E]|D],[R,V]]\rangle \Rightarrow$
$\langle C,\ [D,[replace\_red(M,V,R),[\textbf{val}(V)|E]]]\rangle$
$\langle \overline{\textbf{newind}}; C,\ [D,R]\rangle \Rightarrow \langle C,\ [D,new\_index(R)]\rangle$

In Tables 6–8 we give the definitions generated for the instructions appearing in the above compiler rules.

5.5. Comparison to CAM

Before we can compare our generated abstract machine to the CAM, we have to introduce the CAM in more detail. We restrict the presentation to how variable lookup, λ-abstraction, function application and recursive function definitions are translated to CAM code and discuss the relevant instructions of the CAM. We will use the specifications in [7],[4] where both the CAM and the translation to CAM code are given by natural semantics rules. To avoid having to introduce another notation, we will write these rules in 2BIG notation, although they do not satisfy the static semantics of 2BIG.

5.5.1. Translation of Mini-ML to CAM code

In our generated compiler, we have two stages. First the Mini-ML program is converted into the abstract syntax with access paths. This stage was described by inference rules. Then the actually generated compiler is described by TRS rules and translates a program in abstract syntax into an abstract machine program. These two stages have been intertwined in the inference rules below which specify a translation of Mini-ML programs into CAM code.

[4] Despeyroux's specification slightly differs from [19] and [5]. In the latter the CAM is specified by TRS rules and an ML program is given, which translates Mini-ML programs to CAM code.

In these translation rules the state is an environment, in fact a list of variable names and such lists are constructed as pairs (e, v), where v is a value and e an environment again:

$$\frac{\mathbf{access}(X) \rhd E \to C}{\mathbf{var}(X) \rhd E \to C}$$

Here $\mathbf{access}(X)$ computes given the environment E the access path for the variable X as a sequence of **car** and **cdr** instructions:

$$\frac{V \rhd E \to C}{\lambda X.V \rhd E \to \mathbf{cur}(C)}$$

The variable name is ignored and the expression V is translated into CAM code and used as an argument to the CAM instruction **cur**:

$$\frac{V_1 \rhd E \to C_1 \quad V_2 \rhd E \to C_2}{(V_1\ V_2) \rhd E \to \mathbf{push}; C_1; \mathbf{swap}; C_2; \mathbf{cons}; \mathbf{app}}$$

Here V_1 and V_2 are translated first and the resulting code is combined into a sequence of instructions:

$$\frac{V_1 \rhd (E, X) \to C_1 \quad V_2 \rhd (E, X) \to C_2}{\mathbf{letrec}\ X = V_1\ \mathbf{in}\ V_2\ \mathbf{end} \rhd E \to \mathbf{push}; \mathbf{quote}(R'); \mathbf{cons}; \mathbf{push}; C_1; \mathbf{swap}; \mathbf{rplac}; C_2}$$

Note, that R' is an anonymous variable and, as we will see now when we present the CAM instructions, R' is used by **rplac** to create a cyclic binding.

5.5.2. Instructions of the CAM

The CAM instructions are defined in natural semantics below. In these rules the state is a stack of environments:

$$\frac{}{\mathbf{car} \rhd [(A, B)|S] \to [A|S]} \qquad \frac{}{\mathbf{cdr} \rhd [(A, B)|S] \to [B|S]} \qquad \frac{}{\mathbf{push} \rhd [A|S] \to [A|[A|S]]}$$

$$\frac{}{\mathbf{swap} \rhd [A|[B|S]] \to [B|[A|S]]} \qquad \frac{}{\mathbf{cons} \rhd [A|[B|S]] \to [(A, B)|S]} \qquad \frac{C \rhd [(E, A)|S] \to S_1}{\mathbf{app} \rhd [(clo(C, E), A)|S] \to S_1}$$

$$\frac{}{\mathbf{cur}(C) \rhd [E|S] \to [clo(C, E)|S]} \qquad \frac{}{\mathbf{quote}(R) \rhd [A|S] \to [R|S]} \qquad \frac{V = R'}{\mathbf{rplac} \rhd [(R, V)|[R'|S]] \to [(R, R')|S]}$$

The equality of $V = R'$ means that every occurrence of V in R' is replaced by R'.

5.5.3. Comparison

The following observation will ease the comparison of the rules for the CAM instructions and those for our generated abstract machine instructions. For the CAM the state is a stack of environments, in our generated abstract machine the state has the form $[D, [R, E]]$, where R are redirections and D corresponds to the rest of the stack

Table 9
Comparison to CAM

Generated instruction	CAM instruction
$\overline{\textbf{car}}$	**car**
$\overline{\textbf{cdr}}$	**cdr**
$\overline{\textbf{lambda}}$(V)	**cur**(C)
$\overline{\textbf{equal}}$	**push**[a]
$\overline{\textbf{conv\_11}}$	**swap**
$\overline{\textbf{conv\_12}}$	**cons**[a]
$\overline{\textbf{run}}$	**app**
$\overline{\textbf{conv\_14}}$	**cons**[a]
$\overline{\textbf{letrec}}$	**push**[a]
newind; $\overline{\textbf{conv\_15}}$	**quote**(R'); **cons**; **push**
$\overline{\textbf{conv\_16}}$	**swap**; **rplac**

[a]There is not always a one-to-one correspondence of CAM instructions and generated instructions. For example, $\overline{\textbf{equal}}$ pushes only the environment on top of the stack, whereas $\overline{\textbf{letrec}}$ pushes both the environment and the redirections on top of the stack.

in the CAM and E to the top most element of the stack.[5] Since they have similar effects we get the correspondence of CAM instructions and instructions of the generated abstract machine in Table 9.

As an example, look at the definitions of $\overline{\textbf{lambda}}$ and **cur**:

$$\langle \overline{\textbf{lambda}}(V); C,\ [D,[R,E]]\rangle \Rightarrow \langle C,\ [D,[R,\textbf{clo}(E,\textbf{xlambda}(V))]]\rangle$$

$$\overline{\textbf{cur}(V) \rhd [E|S] \rightarrow [clo(V,E)|S]}$$

Both instructions get the program code V for an expression as an argument and find the environment E on the stack. Then they create a closure on the stack and store the program code V and the environment E in that closure.

The major deviation of the generated instructions from the CAM instructions is the additional handling of redirections. In the above cited specifications of the CAM these are hidden in the meta-language, but when it comes to implementing the CAM in a language like C one has to deal with this problem.

6. Designing other abstract machines

So far, we did experiments with three different languages: SIMP, Mini-ML and action notation. The constructs of these languages include recursive functions and procedures, higher-order functions, local and global variables, assignments, conditionals and loops.

[5] A minor notational observation is, that in the generated abstract machine we use $[h|t]$ to construct environments instead of (t,h).

Mini-Δ [29] is an imperative language with procedures and functions. It allows both call-by-value and call-by-reference parameter passing. Furthermore functions can be passed as parameters. In one of our experiments a benchmark program in the language Mini-Δ was first translated into an action term using an action semantics specification of Mini-Δ. This action term was then executed by interpreting the 2BIG rules for action semantics or by the generated abstract machine for action semantics in C. The abstract machine and the compiler were generated from 100 2BIG rules defining the semantics of 39 action notation constructs including the control, functional, declarative and imperative facet. We did not deal with the communicative facet, nondeterminsm or the interleaving of actions.

7. Conclusions

We presented a new design methodology for abstract machines. In this methodology our semantics-directed generator plays a central role. The generator fully automatically generates compilers *and* abstract machines from semantics specifications of a source language. The generator cannot invent new implementation tricks, so many implementation details of the generated abstract machine depend on the details which have been explicit in the semantics specification. By inspecting the generated abstract machine the developer detects which details should be made more explicit and then changes the semantics accordingly. As an example of this iterative process we show the design of an abstract machine for Mini-ML which turns out to be very similar to the categorial abstract machine.

References

[1] P. Aczel, An introduction to inductive definitions, in: J. Barwise (Ed.), Handbook of Mathematical Logic, North-Holland, Amsterdam, 1977.
[2] H. Aït-Kaci, Warren's Abstract Machine – A Tutorial Reconstruction, MIT Press, Cambridge, MA, 1991.
[3] C. Böschen, C. Fecht, A.V. Hense, R. Wilhelm, An abstract machine for an object-oriented language with top-level classes, Technical Report FB14-No. A 02/94, Computer Science Department, University Saarbrücken, 1994.
[4] L. Cardelli, Compiling a functional language, in: Internat. Symp. on LISP and Functional Programming, 1984.
[5] G. Cousineau, P.-L. Curien, M. Mauny, The categorial abstract machine, in: Proc. FPCA'85, Lecture Notes in Computer Science, Vol. 201, Springer, Berlin, 1985.
[6] L. Damas, R. Milner, Principal type-schemes for functional languages, in: Proc. ACM Symp. on Principles of Programming Languages, 1982.
[7] J. Despeyroux, Proof of translation in natural semantics, in: Proc. Ist Symp. on Logic in Computer Science, LICS'86, Lecture Notes in Computer Science, Vol. 213, Springer, Berlin, 1986.
[8] T. Despeyroux, Executable specification of static semantics, in: Semantics of Data Types, Lecture Notes in Computer Science, Vol. 173, Springer, Berlin, 1984.
[9] S. Diehl, Prolog and typed feature structures: a compiler for parallel computers, Master's Thesis, Worcester Polytechnic Institute, Worcester, Massachusetts, 1993.
[10] S. Diehl, Semantics-directed generation of compilers and abstract machines, Ph.D. Thesis, University Saarbrücken, Germany, 1996. http://www.cs.uni-sb.de/~diehl/phd.html.

[11] S. Diehl, Transformations of evolving algebras, in: Proc. 8th Internat. Conf. on Logic and Computer Science LIRA'97, Novi Sad, Yugoslavia, 1997, pp. 43–50.
[12] S. Diehl, Natural semantics-directed generation of compilers and abstract machines, Formal Aspects of Comput. (1999), to appear.
[13] F.Q.B. da Silva, Towards a formal framework for evaluation of operational semantics, Technical Report ECS-LFCS-90-126, Edinburgh University, 1990.
[14] F.Q.B. da Silva, Correctness proofs of compilers and debuggers: an approach based on structural operational semantics, Ph.D. Thesis, University of Edinburgh, 1992.
[15] J. Hannan, Operational semantics-directed compilers and machine architectures ACM TOPLAS 16 (4) 1994.
[16] J. Jaffar, P.J. Stuckey, S. Michaylov, R.H.C. Yap, An abstract machine for CLP(R), in: PLDI'92, San Francisco, Sigplan Notices, 1992.
[17] T. Johnson, Efficient compilation of lazy evaluation. in: CC'84. Sigplan Notices 19 (6) (1984).
[18] U. Jørring, W.L. Scherlis, Compilers and staging transformations, in: Proc. Thirteenth ACM Symp. on Principles of Programming Languages, St. Petersburg, Florida, ACM, New York, 1986, pp. 86–96.
[19] G. Kahn, Natural semantics, in: Proc. 4th Annual Symp. on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, Vol. 247, Springer, Berlin, 1987, pp. 22–39.
[20] P. Kursawe, How to invent a Prolog machine, in: Proc. 3rd Internat. Conf. on Logic Programming, Lecture Notes in Computer Science, Vol. 225, Springer, Berlin, 1986, pp. 134–148.
[21] P.J. Landin, The mechanical evaluation of expressions, Comput. J. 6 (4) (1964).
[22] X. Leroy, The caml light system, release 0.6 – documentation and user's manual, Technical Report, INRIA, France, 1993.
[23] J.W. Lloyd, Foundations of Logic Programming, 2nd extended ed., Springer, Berlin, 1987.
[24] M. Mauny, A. Suarez, Implementing functional languages in the categorial abstract machine, in: Internat. Conf. on LISP and Functional Programming, 1986.
[25] S. McKeever, A framework for generating compilers from natural semantics specifications, in: P.D. Mosses (Ed.), Proc. 1st Workshop on Action Semantics, BRICS-NS-94-1. University of Aarhus, Denmark, 1994.
[26] M. Mehl, R. Scheidhauer, C. Schulte, An abstract machine for OZ, in: M. Hermenegildo, S.D. Swierstra (Eds.), Proc. 7th Internat. Symp., PLILP'95, Lecture Notes in Computer Science, Vol. 982, Springer, Berlin, 1995.
[27] R. Milner, M. Tofte, R. Harper, The Definition of Standard ML, MIT Press, Cambridge, MA, 1990.
[28] H. Moura, Action notation transformations, Ph.D. Thesis, University of Glasgow, 1993.
[29] H. Moura, D.A. Watt, Action transformations in the ACTRESS compiler generator, in: CC'94, Lecture Notes in Computer Science, Vol. 768, Springer, Berlin, 1994.
[30] A. Mück, Camel: An extension of the categorial abstract machine to compile functional logic programs, in: PLILP'92, Lecture Notes in Computer Science, Vol. 631, Springer, Berlin, 1992.
[31] U. Nilsson, Towards a methodology for the design of abstract machines. J. Logic Programming 16 (1,2) (1993) 163–189.
[32] St. Pemberton, M. Daniels, Pascal Implementation, The P4 Compiler, Ellis Horwood, Chichester, 1982.
[33] D.A. Schmidt, Natural-semantics-based abstract interpretation (preliminary version), in: SAS'95, 1995.
[34] Sun Microsystems, Documentation of the Java Developers Kit-version 1.0 Beta, 1995, available at http://java.sun.com/JDK-beta/index.html.
[35] D.H.D. Warren, Implementing Prolog – compiling predicate logic programs, D.A.I Research Report No. 40, Edinburgh, 1977.